Application Software Cybersecurity Scanning

Lyle Barner
Jet Propulsion Laboratory, California Institue of Technology
lyle.barner@jpl.nasa.gov

Abstract

Scanning software applications for cybersecurity vulnerabilities is a crucial step is assessing the overall health of the application, but how can this kind of scan be performed to give development teams the information they need to make informed design decisions? Two pilot cybersecurity scans were conducted in an attempt to answer this question. A scanning team composed of various subject matter experts was established and worked closely with the development team to perform these scans and capture metrics throughout the process. These interactions and metrics indicate that these scans can be performed in an unobtrusive way and still provide valuable information to development teams regarding the health of their application. This work is not definitive in nature but serves as a foundation for future work.

1. Introduction

Our team's research began with trying to solve a very straight-forward question: how can development teams scan their application for cybersecurity vulnerabilities and use this information to identify risks and inform design decisions? Through two pilot scanning activities, our research team attempted to answer this very question. In this situation, scanning involved performing various types of cybersecurity scans in order to assess the overall cybersecurity health of the application. This information can then be used by development teams to construct a risk profile and inform future design decisions.

In their default configuration, the information provided by many of the scanning tools does not give the user any information about assessing risk or prioritization. Vulnerabilities are returned in the form of raw information such as location, description, and supporting information to help users make their own determination. Users can also import default prioritizations and risk assessments, but there is always a manual component involved in this process. Establishing this process was one of the key aspects of this effort.

The end goal of this activity was to determine how these types of scans should be performed and gather lessons learned along the way. Things like how to assess severity of findings, how to optimize the scanning process, and what teams can expect in terms of resource commitment were all of significant interest to the research team.

This research was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

2. Preparation and overview of scanning process

Preparation for scanning of the target application was a significant part of the energy required for the total process. Constructing a steady framework for performing these scans was necessary to be able to collect the desired metrics and gather sufficient details to perform follow-on activities.

Establishing a team for performing scan was the most immediate need. The scanning team consisted of following four primary roles: facilitator, tools expert, cybersecurity export (CSE), and source code expert (SCE). The facilitator is responsible for managing the activities of the scan. This person makes sure that all the experts have the resources they need to complete the scan and disposition the results. The tools expert is responsible for setting up the tools, making sure they're properly configured, perform the actual scans, and post-processing any of the results if necessary. This individual handles all aspects of running the automated scans. The cybersecurity expert is responsible for reviewing the results of the findings from the automated scans as well as performing the manual scans based on their expert opinion. This will help to catch any items that may have been too complex or outside the scope of the automated scanning tools. Finally, the source code expert is responsible for working with the cybersecurity expert to disposition the results of the automated scans.

Together the source code expert and cybersecurity expert will assess the severity of the vulnerabilities and determine which vulnerabilities require a code change.

One of the most crucial parts of this process was selecting the target application. It was important to find an application with a development team that was dedicated to getting an accurate assessment of the cybersecurity health of their application. Beyond the interest of the teams, it was also necessary to look for software applications that presented the right risk environment to scan for cybersecurity vulnerabilities. Based on this information, the scanning team determined that the appropriate application would likely be a ground software application. Since the process for performing these scans is not well defined, it was important to find projects that were willing to work with the team to capture lessons learned and metrics to define a more formalized process.

From these loose requirements, two target applications were identified for scanning. For the purposes of this paper, these applications will simply be identified as Project 1 and Project 2. The identity of the software applications was something that the scanning team agreed to keep anonymous for security of the applications. These two pieces of software are mature ground applications that are currently being used for mission operations. Both projects are continually being developed to support better operation and new or improved functionality and have a predictable release cycle. Both projects are written in Java and each contains roughly 30k lines of code. Most importantly, these teams wanted to understand the cybersecurity risks present in their application.

Scanning each application was broken down into a manual investigation by a cybersecurity expert, automated scanning of the operating environment for known configuration and third-party software exploits, and automated scanning of the source code for common weakness enumerations (CWEs) using a static analysis tool. [1] The source code scanning was performed using the COTs static code analysis tool Semmle for Project 1 and Semmle and CodeSonar for Project 2. [2][3] For static code analysis scanning the team decided to target only CWEs found within the source code as a in order to measure risk. Semmle supports scanning for 107 different CWEs for Java and CodeSonar supports scanning for 618 different CWEs for Java. Scanning of the operating environment was also performed using Nessus. [4] Nessus performs a scan of the operating environment and searches for common misconfigurations and known vulnerabilities that could be exploited. These automated scans help ensure that maximum coverage is achieved while using minimal resources. This level of coverage is crucial for assessing the cybersecurity risk profile of the application.

Finally, in addition to scanning with the automated tools, manual scanning of the source code and the operating and environment was also performed by a cybersecurity expert. The expert started by using their understanding of the functionality of the application and their own expert opinion to identify potential attack surfaces that could be exploited. Once these attack surfaces were identified, the expert manually reviewed documents and source code to identify any potential vulnerabilities.

Findings from these three types of scans were then reviewed by the cybersecurity expert and source code expert in order to assess the risk presented to the application. Risk was assessed primarily based on the potential consequence of an exploitation. There are many prioritization systems available for assessing CWE risk, but these were not utilized for this activity. Prioritization was assigned at the time of review by the cybersecurity expert and the source code expert. This risk and prioritization information was then used to inform design decisions and code fixes that would be incorporated into upcoming releases.

3. Scanning results

After the installation and setup of the scanning tools was performed each project was scanned using the three scanning approaches described in the previous section. These results were then taken from the tools and post-processed into a common format but remained raw in that no findings were removed.

3.1. Static analysis results

The process for performing the static analysis scanning is fairly straight forward. Once selected, the tool was configured to scan specifically for CWEs. The raw information from the tool was then post-processed into an Excel-based spreadsheet and passed to the source code expert and the cybersecurity expert for review. This spreadsheet contains information about the location of the finding, what CWE it pertains to, and details of the actual concern. During their review, the experts assess the validity of the findings and assign a priority level based on assessment of potential harm to the system if the vulnerability were exploited. The prioritization information is then used to merge cybersecurity specific coding tasks among the other existing coding tasks.

This analysis was based purely on expert opinion without adhering to a formal process. The only goal was to make a determination regarding every finding from the static analysis tool. The team observed the analysis performed by the experts in order to gather information and identify pain points. This data will be used to create a more formalized process as part of follow-on activities. Many of the observations from this process can be found in the lessons learned section.

Ideally, static code analysis should be performed at regular intervals throughout the development lifecycle. This helps to prevent a large backlog of vulnerabilities as development proceeds. Both of the projects scanned during the course of the pilot were very mature and had been performing these regular scans, but CWE checks had not been enabled. The recommended course of action is simply to enable CWE checks as a regular part of the static code analysis that is already being performed. While these particular scans were performed outside of the typical scanning cycle, it is easy to see how they could be incorporated.

The results of the static analysis scans of Project 1 and Project 2 are shown in the tables and figures below. These tables and figures contain information about the raw results as identified by the static code analysis tools. Full, detailed results will not be presented in this paper for the sake of brevity. Tables 1 and 2 show the distribution of CWE findings identified by Semmle. Table 3 shows the results of a CodeSonar scan that was performed on Project 2 outside of the regular scanning process due to license availability issues.

Table 1: Results of Semmle scan of Project 1

CWE	Description	Count
CWE-022	Improper limitation of a pathname to a restricted directory (‘path traversal’)	35
CWE-190	Integer overflow wraparound	3
CWE-129	Improper validation of array index	1

Table 2: Results of Semmle scan of Project 2

CWE	Description	Count
CWE-089		48
CWE-022	Improper limitation of a pathname to a restricted directory (‘path traversal’)	11
CWE-190	Integer overflow wraparound	7
CWE-311	Missing encryption of sensitive data	3
CWE-597	Use of wrong operator in string comparison	3
CWE-129	Improper validation of array index	2

Table 3: Results of CodeSonar scan of Project 2

CWE	Description	Count
CWE-252	Unchecked return value	6
CWE-374	Passing mutable objects to an untrusted method	17
CWE-391	Unchecked error condition	426
CWE-396	Declaration of catch for generic exception	977
CWE-397	Declaration of throws for generic exception	677
CWE-398	7PK code quality	2
CWE-456	Missing initialization of a variable	1
CWE-459	Incomplete cleanup	400
CWE-476	NULL pointer dereference	12
CWE-478	Missing default case in a switch statement	16
CWE-484	Omitted break statement in switch	17
CWE-500	Public static field not marked final	2
CWE-543	Use of singleton pattern without synchronization in a multithread context	5
CWE-561	Dead code	28
CWE-563	Assignment to variable without use	57
CWE-569	Expression issues	1
CWE-571	Expression is always true	2
CWE-581	Object model violation: just one of equals and hashcode defined	4
CWE-595	Comparison of object references instead of object code	25
CWE-596	Incorrect semantic object comparison	1
CWE-597	Use of wrong operator in string comparison	4
CWE-607	Public static final field references mutable object	2
CWE-662	Improper synchronization	2
CWE-681	Incorrect conversion between numeric types	6
CWE-682	Incorrect calculation	4
CWE-767	Access to critical private variable via public method	13
CWE-775	Missing release of file descriptor or handle after effective lifetime	2
CWE-845	CERT Java secure coding section 00 – input validation and data sanitization	51
CWE-846	CERT Java secure coding	7

	section 01 – declarations and initializations	
CWE-849	CERT Java secure coding section 04 – object orientation	11
CWE-851	CERT Java secure coding section 06 – exceptional behavior	16
CWE-854	CERT Java secure coding section 09 – thread APIs	1

In addition to determining the distribution of findings among the different CWEs, post-processing was performed to determine the density of the findings among the source code modules. The results of the post-processing are shown below in Figure 1 and Figure 2. In order to protect the identity of the projects, the names of the modules will remain anonymous.

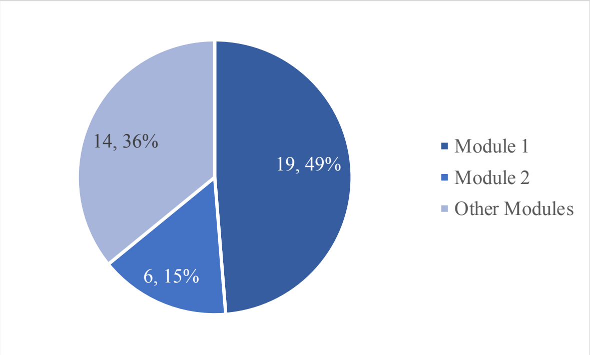


Figure 1: Distribution of findings among source code, Project 1

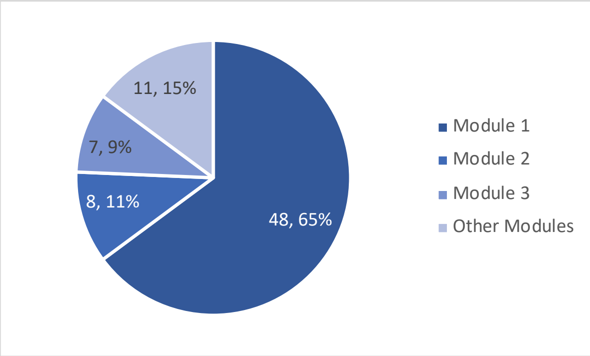


Figure 2: Distribution of findings among source code modules, Project 2

The results shown in Table 2 and Table 3 are from the same source code from Project 2, but different static analysis tools. The primary reason for the much larger number of findings shown in Table 2 is that CodeSonar supports a much larger number of CWEs than Semmle. A comparison between the results from Semmle and CodeSonar can be seen below in Table 4.

Table 4: CodeSonar and Semmle comparison

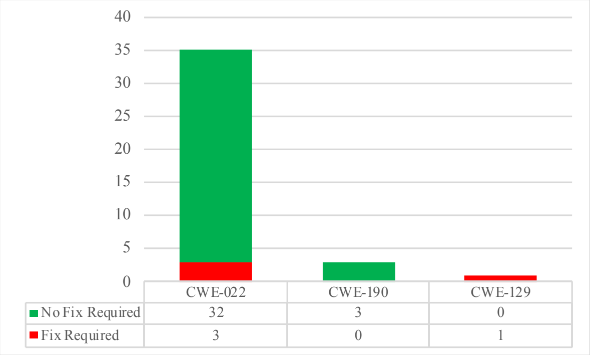
	Semmle	CodeSonar
Covered CWEs	107	618
Total findings	74	2795
Finding concurrence	1 instance of partial concurrence	

Of the various CWEs that are covered by both Semmle and CodeSonar, there are 81 CWEs that are covered by both tools. Of these 81 common CWEs there were 17 instance of no concurrence, 63 instances of potential concurrence, and 1 instance of partial concurrence. No concurrence is defined as no matching findings for a given CWE. Potential concurrence is defined as instances where no findings were identified by either tool, so it is not possible to make a definitive judgement on the overlap between the tools. Finally, partial concurrence is defined as an overlap of at least one finding for a given CWE. In this particular analysis, there was only 1 individual finding that was identified by both tools.

After the scans were performed, the findings were analyzed by the cybersecurity expert and the source code expert to determine the validity of the finding and the risk associated with the finding. The risk assessment is based purely on the opinion of the cybersecurity expert and the source code expert. Assessing risk requires detailed knowledge of the target application, operating environment, and concept of operations of the system as a whole. A more formalized risk categorization process is currently being developed to assist teams in streamlining the process for assessing risk.

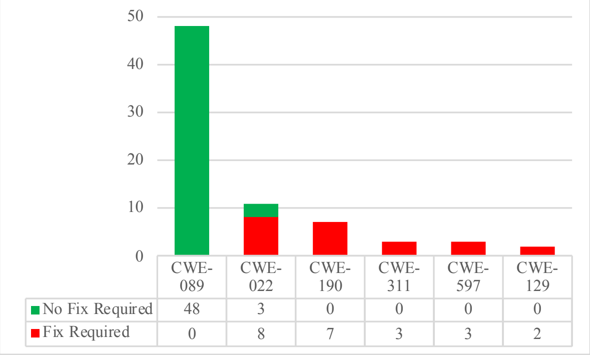
If this risk is sufficiently high, a code change will be implemented, and design rules can be cataloged to help guide future development. Performing a static analysis scan of the code base before the peer review helps to focus the energy of the peer reviewers. The automated scan performed by the static analysis tool ensures 100% coverage of the source code and the expert review of the results ensures that the dev team is provided with actionable information. The result of this process is a set of prioritized vulnerabilities that include a high-quality description of the vulnerability and potentially even details on a fix. These code fixes can then be inserted directly into the existing development lifecycle along with other coding tasks.

Figure 3 and Figure 4 below show the number of findings that were deemed to be valid for each project. This activity was only performed for the findings identified via Semmle. The CodeSonar results were not generated in time to be included in the metrics collection process.



	CWE-022	CWE-190	CWE-129
🟩 No Fix Required	32	3	0
🟥 Fix Required	3	0	1

Figure 3: Findings validity for Project 1



	CWE-089	CWE-022	CWE-190	CWE-311	CWE-597	CWE-129
🟩 No Fix Required	48	3	0	0	0	0
🟥 Fix Required	0	8	7	3	3	2

Figure 4: Findings validity for Project 2

Conveniently, the static analysis tools provided all of the information that is needed to make a risk assessment for a given vulnerability. For example, the Semmle results provide users with the exact line of code that is of concern, a description of the problem, and even a generic discussion of how this type of vulnerability is usually addressed. The only difficulty associated with making a determination is marrying this data up with other controls that may or may not exist in the operating environment. This is where the knowledge of the source code expert is crucial.

3.2. Expert review and Nessus scan results

The process for performing a scan of the operating environment was quite similar to the process for performing the static analysis scan. First, the tools and documents of interest were identified. Once this information was obtained, the automated scan was performed and the manual review by the cybersecurity expert was also conducted. This information was then manually organized and reviewed by the cybersecurity expert and source code expert simultaneously. Finally, the experts made a determination regarding every finding and assigned a priority level based on expert opinion. Again, the process for performing the scan of the operating environment was left intentionally vague and was reviewed closely by the team to capture lessons learned for follow on activities.

Similar to the static code analysis scanning, the operating environment should ideally be scanned at regular intervals during the development lifecycle. Even if there are no changes to the third-party dependencies of the target application, new vulnerabilities can arise over time. In many projects this may or may not be how this type of scanning is performed and may introduce additional overhead to the development team. For these particular projects, this type of scan had been performed in the past, but not on a regular basis.

The products of the Nessus scan and the manual expert review were much less formal than the static analysis results. In many cases, these kinds of findings did not generate changes to the source beyond what was already identified by the static analysis tools. Instead, these findings can be flowed into changes to the configuration of the operating environment and procedural changes associated with using the applications. Additionally, these types of scans did not generate the same number of findings that were uncovered by static analysis of the source code. For these reasons, only a subset of the results will be shared.

Scanning by the cybersecurity expert revealed several instances of uncontrolled information contained within the application documentation. For example, some documents contained admin login credentials. This poses a significant risk if used for malicious purposes. Scanning using Nessus also revealed configuration issues with Apache Tomcat serve being used by the applications that would leave the application susceptible to JDK vulnerabilities.

After the results of the scanning were compiled into a final report, the cyber security expert is able to present the findings to the source code expert for review. Due to the nature of the scan, there is a much lower false positive rate, so the primary job of the source code expert is to simply understand the concerns, and a much lower false positive rate is to be expected. The results of the Nessus scan may not capture the all of security controls that are in place on the system, so this may be where there could be some false positives. Using the details of the security controls present to tailor the results proved to be the most time-consuming part of the process, though the cost associated with this activity is still reasonable. The cybersecurity expert must take time to understand these controls to better tailor their analysis. Again, risk is assessed based on the expert opinion of the cybersecurity expert. Follow-on activities will work to determine a more formalized process for capturing risk assessments.

3.3. Metrics collected during analysis

During the course of the scanning, analysis, and dispositioning process the team collected metrics to better inform future activities. Metrics of concern are primarily analysis time and metrics distribution per thousand lines of code, but other metrics were collected along the way as well. The metrics collected during these pilot activities are provided below in Table 5 and Table 6.

As can be seen the following Tables, there was very little cost associated with performing these types of scans. For two mature software projects, it took less than 15 hours to perform the scans and disposition the results. The analysis cost can be broken down into three major categories: tools setup, source code expert analysis, and cybersecurity expert analysis. Tools setup encompasses setting up the analysis tools, running the analysis, and post-processing the results. For Project 2, the slightly higher number of results prompted an additional analysis activity to have the software lead organize the analysis into larger categories for the source code expert and cybersecurity expert to review. Nearly all of these analysis categories can see improvement with a more well-defined tool deployment process and a more formalized review process. A low-cost process would mean that teams can easily identify and rectify cybersecurity issue and improve the overall security of their codebase without being overly concerned with cost.

While this does not include the time needed to make the necessary code fixes, it still encompasses a large portion of the cost. It shows that this type of scan can be readily performed without having to dedicate significant resources to the effort. More pilot activities are needed to better determine the costs associated with performing this type of scan, but the initial results are promising.

Table 5: Project 1 analysis metrics

Metric	Value
Language	Java
Lines of code	24,310
Total findings	39
Finding rate	1.61/thousand lines of code
True defect rate	0.16/thousand lines of code
Findings requiring a code change	12.5%
Analysis time	Tools setup: 5-6 hrs SCE analysis: 2-3 hrs CSE analysis: 2 hrs Total: 9-11 hrs

Table 6: Project 2 analysis metrics

Metric	Value
Language	Java
Lines of code	36,725
Total findings	74
Finding rate	2.01/thousand lines of code
True defect rate	1.39/thousand lines of code
Findings requiring a code change	69%
Analysis time	Tools setup: 2-3 hrs Lead analysis: 2-3 hrs SCE analysis: 4-5 hrs CSE analysis: 4 hrs Total: 12-15 hrs

3.4. Observations based on analysis results

There were many pieces of interesting information that can be gathered from the raw analysis data and the dispositioning results. Much of this information flows into the lessons learned that are captured in the following section, but there is still some information that is worth mentioning here.

The first piece of information that is apparent is that these scans can be performed in a relatively short amount of time, even for mature software applications. Both pilot activities were completed with the risk assessment portion of the process in under 15 hours. This is manageable for most projects, but more scans are needed to gather more data points.

The team also discovered that the analysis time does not necessarily scale with the number of findings that are uncovered. While more warnings generally could mean that the dispositioning process could take longer to complete, this is not always the case. For example, imagine that more warnings are discovered in third application, but nearly all the warnings are related to sanitization of user input. This type of finding is usually dispositioned fairly quickly and may not increase the dispositioning time noticeably. However, this will likely increase the time required to implement the code fixes associated with the true defects.

4. Conclusions and lessons learned

At the conclusion of each scanning process the team convened to discuss the lessons learned and what could potentially be improved for future efforts. There were many lessons learned as part of these pilot activities, but they roughly follow the different phases of the scanning process from setup and installation though disposition.

4.1. Setup lessons learned

There is relatively little cost associated with preparing to perform these kinds of scans. The metrics collected during these pilot activities showed that the total time to perform analysis was no more than 15 hours. Of this total time, it took less than 3 hours for a knowledgeable user to set up the various automated scanning tools and another 2-3 hours to create the associated post-processing scripts. This cost can further be reduced as the deployment process is refined through follow on activities.

When scanning the source code specifically for cybersecurity vulnerabilities, configuration of the scanning tools is very important. While these tools often have utility beyond just scanning for cybersecurity vulnerabilities, limiting the functionality can be helpful to focus teams. Static analysis tools can often generate a lot of false positives which can be time consuming to disposition. Adding additional noise through unnecessary queries can make this an even bigger problem. Make sure to sit down with the team to determine what are the most effective queries to be run for the given analysis. Another option for solving this issue is to create segregated dashboards so that queries are effectively grouped. The best option would likely be to have a portion of the development team that is dedicated to performing these scans, analyzing the results, establishing the risk, and distributing this information to other members of the team as necessary.

There is very little overlap between the different static code analyzers when comparing results against matching CWEs. It is possible to conclude from this finding that to truly get the best results users should run as many static code analyzers as possible for a given analysis to get the best coverage of the CWEs. This of course means that more output will be generated and the analysis time will be increased. To mitigate this teams should perform these scans regularly and address the findings frequently to ensure that the process does not become unmanageable.

The source code, operating environment, and associated documentation must all be examined as part of the scanning process. There are different pieces of information contained in each one of these domains. It is not sufficient to examine only one area. Additionally, there are some vulnerabilities that only present themselves through a combination of more minor warnings that are distributed among the different domains.

4.2. Scanning and post-processing lessons learned

Most of the information needed to perform this type of analysis is readily available. The scanning tools provide users with most of the necessary information for diagnosing vulnerabilities within their codebase. The risk assessment/prioritization process is the missing piece that must be addressed in follow-on activities.

Due to the sensitive nature of the information contained in the results, special measures must be taken when sharing the results. The raw results that are being shared contain information that would potentially be very valuable to an attacker. Because of this it makes sense to keep the results in a segregated area that is protected from more general information about the codebase. This area should only be accessible to the minimal set of developers required. Protecting the results of the scans means that risk items are protected and vulnerabilities are not revealed to a larger audience.

The number of vulnerabilities that exist within the codebase are not based on the number of lines of code, but instead on the functionality of the code. It might be reasonable to expect that doubling the amount of code being scanned would result in double the amount of vulnerabilities uncovered, but this is not the case. However, for example, if an application has more opportunities for user input, or makes use of a network connection, this codebase will likely have more vulnerabilities than a similarly sized application with less user input or no interface with a network connection. This can make it extremely difficult to estimate the actual effort associated with performing this kind of cybersecurity scan. Again, the recommendations from this lesson learned is to perform scans regularly so as not to create a large backlog of findings.

The vulnerabilities are often not evenly distributed throughout the code. There will be many modules that have very few or even now cybersecurity vulnerabilities. Again, this is primarily due to the architecture and the functionality of the application. For instance, if there is a particular module in the application that is responsible for handling user input this module is likely to have a higher density of the cybersecurity vulnerabilities than other modules in the application. If a particular module has a high distribution of vulnerabilities, this means it may be a good target for further analysis. It could also mean that the team may need consider refactoring the module to better protect the application.

4.3. Dispositioning lessons learned

Grouping warnings into different categories is helpful for the dispositioning process. Some scanning tools can handle this step automatically, but if not, this is something to be considered. As previously mentioned, these scanning tools can often generate a large amount of output. This step can help break this output down into more manageable tasks that can be distributed to different members of the development team. Ultimately this helps to streamline the process of assessing the risk associated with a finding or set of findings.

The cybersecurity expert and the source code expert must work together in order to make a determination about the validity of finding. Each finding from the scanning tools can be examined from the perspective of the cybersecurity expert as well as the perspective of the source code expert. The cybersecurity expert may have information about the finding that can better explain its importance to the source code developer. Likewise, the source code developer may have information about the operation of the application that negates the importance of the finding. Essentially, they must work together as a team to come to agreement about the severity of a given finding. This is true for determining the risk associated with a given finding. If the common risk assessment technique is applied (likelihood vs. consequence) the source code expert can assist in determining the likelihood of a finding being exploited, while the cybersecurity expert can speak to the consequence of the vulnerability being exploited

True positives can be used to create a rolling list of design rules. As vulnerabilities are uncovered in the code and fixes are implemented, a database of design rules can be formulated. These design rules and patterns can help to reduce the number of vulnerabilities that are introduced during regular development activities.

Code fixes are related to the type of finding, more so than the individual finding. For instance, let’s assume that the scanning tools flag multiple findings in the source code that there is potential for privilege escalation. However, what the scanning tools are unaware of are any outside protections that would prevent this kind of escalation. So, in this instance, none of these findings would be considered to be valid and it is possible to disposition them all at the same time. There are some instances where this is not the case, but during our pilot activities most categories of findings followed this pattern. This information can be used to streamline the disposition and risk assessment process.

A detailed process for prioritizing and assessing risk associated with vulnerabilities must be created. This area is where the majority of the analysis time was spent. Creating a formalized process will drive down the overall analysis time and is something that will be addressed in future work.

5. Future work and extensions

This pilot activity was conducted with the intention of leading into many follow activities. Some of these activities were known before starting the pilot scans and helped to shape how the scans were conducted, but more of these activities were discovered during the course of the pilot.

The primary follow-on activity is to define a formalized process for performing these types of cybersecurity scans. The metrics collected as a part of the original scans will be used to help inform this process. This will likely be an iterative activity that will involve scanning more projects and further refining the process. The formalized process is important for ensure teams are performing these scans in a way that is consistent and repeatable. The team plans to continue collecting metrics on any additional scans that occur in order to better understand the relationship between lines of code, number of findings, and overall time to complete the analysis. This information can be very helpful when teams have very little budget to use and would like to better understand the commitment that is required to perform this kind of scan.

In addition to the more formalized process, the scanning team determined that training and exposure materials to publicize this activity internally would also be helpful. Making development teams aware that this type of scan Is necessary to perform is beneficial on its own, but this activity has the added benefit of potentially identifying early adopters that would be willing to participate in the process of refining the scanning process. This activity also allows teams to voice their concerns in a more generic way so that the team has a chance to address them either in the formalized process or in future exposure materials or training activities. The end goal is to provide teams with a way to incorporate this information into their development lifecycle by providing them with information about what aspects of the lifecycle this kind of scan affects.

One of the most surprising results from this pilot activity was the lack of overlap between Semmle and CodeSonar when comparing the results of the CWE scans. This begs the obvious question of why the results are so different. Determining which tools provide the best results will be important for maximizing the return on investment for implement these kinds of scans. The first step in solving this problem would be to performing a benchmarking activity. This could be done by running both analyzers against a source code test suite that contains known vulnerabilities and then comparing the results of the tools to see how many false positives are identified and how many true positives are identified. The are several open source test suites that could be used for this purpose. For example, the Juliet test suite is a collection of artificially injected CWEs for both Java and C that could be used to perform such a comparison. This information could then lay the foundation for future work to judge the effectiveness of different static code analysis tools for performing cybersecurity scans.

During the process of analyzing the data from the various scans the team had some issues managing all the sources of information. There is an internally available tool called SCRUB that could potentially be a good solution to this problem. SCRUB is a peer review tool that is used to run static analysis tools, post-process the results, and aggregate the results into a centralized location for review. [7] Users can also manually input findings into the SCRUB results if something is discovered that was no identified by the automated tools. SCRUB can also be used to link users back to the source code so they are able to review the code to make a determination on a particular finding regarding the risk that it presents to the project. SCRUB also has the added benefit of allowing multiple users to review the code and findings simultaneously and then vote on their resolution. In this situation, SCRUB could be used by the scanning team to perform automated scans, collect manual inputs from cybersecurity experts, and then allow source developers and cybersecurity experts to review findings independently while still gathering input from both sides. This approach serves to streamline the review process by focusing on only the findings that have been deemed worthy of debate when making design decisions. Finally, since SCRUB is already being used to drive peer reviews, this information could easily be rolled in to the existing process to minimize the impact to development teams.

One of the primary pieces of feedback the team received from the developers on the pilot projects was that it would be really helpful to have some kind of mechanism for prioritization of the findings from the static analysis tools. There are many different methods of prioritizing these warnings. MITRE maintains several different prioritization techniques as part of their CWE list that can be used to ranking the different findings. Each of these rankings are slightly different and have different underlying methodologies. NASA also maintains a yearly ranking of what they consider to be the top 25 CWEs for both ground and flight software. This activity should examine the applicability of these ranking systems and attempt to make a recommendation as to which ranking system makes the most sense for these types of scans. Creating an effective prioritization system will help to streamline the process for assigning risk to a given finding or set of findings.

6. Conclusion

At the start of this activity the scanning team was attempting to gather information about how to best incorporate cybersecurity scanning into the development lifecycle in a way that gives teams the information they need about risks in order to make informed design decisions.

These scans were performed with a relatively small amount of effort given the maturity of the projects being scanned. This indicates that scanning a well-established project and assessing the risk of the vulnerabilities discovered is something that could be done relatively easily by most teams. The necessary information to assess risk is commonly readily available and can be combined with the information provided by the scanning tools to make an effective determination. Beyond the metrics collected, there were many lessons learned from this pilot process regarding how to best perform these kinds of scans and present this information to teams for decision making purposes.

Future work is needed to better understand the best way to implement these scans on a large scale. Creating a formalized process that is able merge into existing process will require further research, but these pilot scans have served their purpose as a proof of concept. Benchmarking the different static analysis tools is likely a required step before rolling out these kinds of scans on a large scale. The use of facilitating tools could help to streamline the review and risk assessment process. A prioritization methodology would also be helpful for objectively assessing risk and presenting teams with consistent data for making design decisions.

Ultimately, this work will be continued with the end goal of creating a formal process and a stable set of tools for performing these kinds of scans

7. References

[1] MITRE, “About CWE”, Website, MITRE, March 2018, Retrieved from https://cwe.mitre.org/about/index.html.

[2] Semmle, “Engineering Analytics”, Website, Semmle, 2018, Retrieved from https://semmle.com/products/.

[3] GrammaTech, “Product Datasheet”, Website, GrammaTech, 2018, Retrieved from https://www.grammatech.com/sites/default/files/codesonar-datasheet.pdf.

[4] Tenable, “Nessus Professional”, Website, Tenable, Dec 2017, Retrieved from http://info.tenable.com/rs/934-XQB-568/images/NessusPro__DS__EN_v8.pdf.

[5] Holzmann, G.J., “SCRUB: a tool for code reviews”, Website, Jet Propulsion Laboratory, 2009, Retrieved from http://spinroot.com/gerard/pdf/ScrubPaper_r